Research report : Collaborative Peer 2 Peer Edition: Avoiding Conflicts is Better than Solving Conflicts

S. Martin and D. Lugiez
LIF UMR 6166 Aix-Marseille Université CNRS

ABSTRACT

Collaborative edition is achieved by distinct sites that work independently on (a copy of) a shared document. Conflicts may arise during this process and must be solved by the collaborative editor. In pure Peer to Peer collaborative editing, no centralization nor locks nor time-stamps are used which make conflict resolution difficult. We propose an algorithm which relies on the notion or semantics dependence and avoids the need of any integration transformation to solve conflicts. Furthermore, it doesn't use any history file recording operations performed since starting the edition process. We show how to define editing operations for semi-structured documents i.e. XML-like trees, that are enriched with informations derived for free from the editing process. Then we define the semantics dependence relation required by the algorithm and we present preliminary results obtained by a prototype implementation.

1 Introduction

Collaborative edition becomes more and more popular (writing article with SVN, setting appointments with doodle, Wikipedia articles,...) and it is achieved by distinct sites that work independently on (a copy of) a shared document. Several systems have been designed to achieved this task but most of them use centralization and locks or weak centralization via time-stamps. A alternative approach is the Peer to Peer approach -P2P in short- where new sites can freely join the process and no central site is required to coordinate the work. This solution is more secure and scalable since the lack of central site prevents from failures and allows for a huge number of participants. In this paper we focus on editing semi-structured documents, called XML trees from now on, using the basic editing operations *add, delete* for edges or *changing labels* in the document. Since the process is concurrent, conflicts can occur: for instance a site s_1 changes the label $Introduction$ of an edge by $Definition$ when another site s_2 want to relabel $Introduction$ by $Abstract$. Then s_1 informs s_2 of the operation performed and conversely. Executing the corresponding operations leads to an incoherent state since the sites nor longer have identical copies of the shared document. In the optimistic P2P approach, each operation is accounting for and conflicts are solved by replacing the execution of an operation op_2 performed concurrently with op_1 by $IT(op_2, op_1)$ where IT is an integration transformation defined on the set of operations. This transformation computes the effect of the execution of op_1 on op_2, i.e. the *dependence of* op_2 *from* op_1.

In the word case, the transformations proposed in [12, 3, 8, 10, 13] turned out to be non-convergent, see [7] for counter-examples. In particular, none of these transformations satisfy both properties $TP1$ (a local confluence property) and $TP2$ (integration stability) that are sufficient to ensure convergence [12]. Currently, no convergent algorithm based on the integration transformation is known for words. For XML trees, algorithms and operations have been proposed (like in [1]), but they have the same problem as in the word case or use time-stamps (see [11]) i.e. are not true P2P.

We propose a new algorithm that relies on *semantic dependence* of operations which allows to reduce the integration transformation to a trivial one: $IT(op_2, op_1) = op_2$. This is possible since we enrich the data structure by adding informations coming for free from the editing process on trees yielding an important property: each edge is uniquely labelled. Furthermore labels also record the level of dependence of the sites that created or modified them. These properties allow to get a simple convergent editing algorithm which doesn't require any history file recording all operations done since the beginning of the edition process. Since a word can be encoded as a tree, this algorithm also solves the word case, at the price of a more complex representation. These ideas have been implemented in a prototype that proved that the editing is done efficiently and that the process is scalable.

Section 2 discusses the current approaches to collaborative editing, and we present our editing algorithm in section 3. The data structure used for XML trees is described in section 4 and our first results are given in section 5. Missing proofs can be found in the full research report.

2 Related Works

Many collaborative edition framework have been proposed, and we discuss only the most prominent ones.

Document synchronization framework. *IceCube* (see [9]) is a operational-based generic approach for reconciliating divergent copies. Conflicts are solved on a selected site using optimization techniques relying on semantic static constraints (generated by document rules) and dynamic (generated by the current state of the document). Complexity is NP-hard and this approach is not a true P2P solution (each conflict is solved by one site). The *Harmony* project [4] is a state-based generic framework for merging two divergent copies of documents. These documents are tree-like data structure similar to the unordered trees that we discuss in section 4. The synchronization process exploits XML-schema information and is proved terminating and convergent for two sites.

Integration transformation based framework. *So6* [11] is a generic framework based on the *Soct4* algorithm which requires the local confluence property (TP1). It relies on continuous global order information delivered by a times-tamper, which is not pure P2P since it relies on a central server for delivering these time-stamps. The *Goto* system (Sun et al.[14]), or *SDT* (Du Li and Rui Li [2]) rely on forward and backward transformation (for undoing operations). These algorithms need to reorder the history of operations which involve a lot of computations to update the current state in order to ensure convergence.

Goto (Sun et al. [14]), *Adopted* (Ressel et al. [12]) and *SDT* (Du Li and Rui Li [2]) rely on the local confluence property ($TP1$) and on the integration stability property ($TP2$) to guarantee convergence. A main issue is to ensure that operation integration takes place in the same context and return the same result and each algorithm has its own solution. For instance, *Goto* uses a forward (IT) and a backward ET) transformation to reorder the history (record of all operations performed). *Adopted* computes the sequence of integrations as a path in a multi-dimensional cube. The main drawback of these approach is that it is hard to design set of useful operations and integration transformations that satisfy both $TP1$ and $TP2$. For instance, no such set exists in the word case nor for linearly ordered structures.

The set of operations given by Davis and Sun provides operations on trees for the Grove editor [1], but this set doesn't satisfy the local confluence property TP1. Therefore, there is little hope to get a convergent editing process. *OpTree* [5, 6] present a framework for editing trees and graphical documents using *Opt* or the *Soct2*, and relies extensively on history files containing all operations performed on the date. The complexity is at least quadratic in the size of the log file and no formal proof of correctness is given.

A main problem of all these solutions -even when convergence is guaranteed- is that they rely on manipulation of history files that records all operations performed and these computations can become quite expensive.

3 Conflict-free Solution

We propose a generic schema for collaborative editing which avoid the pitfalls of previous works by avoiding the need to solve conflicts. First we give an abstract presentation of this editing process and of the properties required to ensure its correctness, then we show how it works for XML trees.

Each site participating to the editing process executes the same algorithm (given in figure 1) and performs operations on his copy of the shared documents. Operations belong to a set of operations Op , and we assume that there is a partial order $\succ_s$ (i.e. an irreflexive, antisymmetric, transitive relation) on operations and we write $op_1 \parallel_s op_2$ iff $op_1 \not\succ_s op_2$ and $op_2 \not\succ_s op_1$. This ordering expresses causal dependencies of the editing process: $op_1 \succ_s op_2$ iff op_2 depends from op_1 (for instance op_1 creates an edge and op_2 relabels this edge). In our model the set $OpDep$ as $op \in Op, \forall op' \in OpDep | op \succ_s op'$ is bounded set. We show how to compute this relation for XML trees in section 4.3. A sequence of operations is denoted by $[op_1;\ldots;op_n]$ and the result of applying op_1, followed by $op_2, \ldots, op_n$ to the document t is denoted by $[op_1;\ldots;op_n](t)$. The set of operations $(Op, \succ_s)$ is *independent* iff $\forall op, op' \in Op \ \ \forall t, op \parallel_s op' \implies [op, op'](t) = [op', op](t)$.

A sequence $[op_1;\ldots;op_n]$ is valid if for all op_i, op_j occurring in the sequence, $op_i \succ_s op_j$ implies $i < j$. In other words, the sequence is *a linearization* of the partial order defined by $\succ_s$ on the set $\{op_1, \ldots, op_n\}$. Given a valid sequence $[op_1;\ldots;op_n]$, a substitution σ of $\{1, \ldots, n\}$ is compliant with $\succ_s$ iff the sequence $[op_{\sigma(1)};\ldots;op_{\sigma(n)}]$ is valid. This yields that $op_i \parallel_s op_j$ iff $op_{\sigma(i)} \parallel_s op_{\sigma(j)}$ or in other terms, σ doesn't change the causality relation between operations. The collaborative editing algorithm that we propose relies on the following proposition[1]:

Proposition 1 *Let $(Op, \succ_s)$ an independent set of operations. Let $[op_1, \ldots, op_n]$ be a valid sequence of operations in Op and let σ be a substitution compliant with $\succ_s$. Then $[op_1, \ldots, op_n](t) = [op_{\sigma(1)}, \ldots, op_{\sigma(n)}](t)$*

PROOF. Firstly, we prove that exchanging two consecutive non-dependent operations doesn't change the result.

Let τ_i the substitution such that $\tau_i(i) = i+1, \tau_i(i+1) = i$ and $\tau_i(k) = k$ otherwise. Let $[op_1;\ldots;op_n]$ be a valid sequence and let $op_i \parallel op_{i+1}$. We prove that $[op_1;\ldots;op_n](t) = [op_{\tau_i(1)};\ldots;op_{\tau_i(n)}(t)]$ as follows:

$$\begin{aligned}
[op_{\tau_i(1)};\ldots;op_{\tau_i(n)}](t) &= [op_1;\ldots;op_{i-1};op_{i+1};op_i;op_{i+2}\ldots;op_n](t) \\
&= [op_{i+1};op_i;op_{i+2}\ldots,op_n](t') \text{ with } s' = [op_1;\ldots,op_{i-1}](t) \\
&= [op_i;op_{i+1};op_{i+2}\ldots,op_n](t') \text{ since } (Op, \succ_s) \text{ is independent} \\
&= [op_1,\ldots,op_n](t)
\end{aligned}$$

Secondly we prove the result by induction on the number of elements in the sequence $[op_1;\ldots;op_n]$.

- Base case: $n = 1$ straightforward.
- Induction case: Let $[op_1;\ldots;op_n]$ be a valid sequence of Op .

  Let $[op_{\sigma(1)};\ldots;op_{\sigma(n)}]$ be another linearization of $\{op_1,\ldots,op_n\}$.

  We prove that $[op_1;\ldots;op_n](t) = [op_{\sigma(1)};\ldots;op_{\sigma(n)}](t)$.

  By definition op_1 is a maximal element of $\succ_s$. This element occurs at position j in $l = [op_{\sigma(1)};\ldots;op_{\sigma(n)}](t)$. Let τ_k be the subtitution that exchanges the elements of l at positions k and $k+1$ and leaves other elements unchanged.

  Since op_1 is maximal, any operation op' occurring in l at position $k < j$ is such that $op' \parallel op$.

  Therefore there is a sequence $\tau_{j-1},\ldots,\tau_1$ of substitutions such that the application of these substitutions to $[op_{\sigma(1)};\ldots;op_{\sigma(n)}]$ yields a sequence $[op_1;op'_2;\ldots;op'_n]$ such that (i) $[op_1;op'_2;\ldots;op'_n](t) = [op_{\sigma(1)};\ldots;op_{\sigma(n)}](t)$ (by our first result) and (ii) $[op_1;op'_2;\ldots;op'_n]$ is a linearization of $op_1,\ldots;op_n$.

  Therefore $[op'_2;\ldots;op'_n]$ is a linearization of $op_2,\ldots,op_n$.

  By induction hypothesis, we get $[op'_2;\ldots;op'_n](t') = [op_2,\ldots,op_n](t')$.

  Taking $s' = [op_1](t)$ yields the result.

□

Another statement of the proposition is that the execution of any linearization of a partial order on some initial value yields the same result.

[1]This result is a classical result in the field of partial order

The dependenceOf function. In our setting, operations are issued by sites and are numbered with an operation number on this site. For instance, to delete a node in a tree, the operation is defined by the action *delete*, the site identifier *SiteId* of the site which issues this deletion and the operation number *OpCount* on this site. Furthermore, the data structure (the shared document) is build using these operations and stores this information for each component (nodes or edges for trees for instance). A request r is a triple composed of an operation *op*, a site identifier *SiteId*, and an operation number *OpCount*. We assume that there is an function $dependenceOf(r)$ which returns for each request r, the pair $(SiteId' : OpCount')$ of any operation op' such that $op' \succ_s op$. Actually, this operation can return such pairs only for the minimal (ofr $\succ_s$) operations op' such that $op' \succ_s op$. In section **??**, we show how to define effectively and in a simple way this function for XML trees.

The (Fast Collaborative Editing) FCeditAlgorithm. The procedures (except *Main()*) of the generic distributed algorithm *FCedit* are given in figure 1. Each site has an unique identification stored in *SiteId*, a operation numbering stored in *Opcount*, a copy of the document t and a list *WaitingList* of requests awaiting to be treated. The function *dependenceOf(r)* with $r = (op, SiteId : OpCount)$ returns the pairs $(nSite : cSite)$ with *nSite* a site identifier, *cSite* some operation count, such that op depends from an operation issued from site *nSite* with operation count *cSite*. This function is defined simultaneously with the data structure, set of operations and dependence relation, see section 4.3 for the definition used for XML-trees. The *Main()* procedure (not given in figure 1) calls *Initialize()* and enters a loop which terminates when the editing process stops. In the loop, the algorithm choose non-deterministically to set the variable *op* to some user's input and to execute *GenerateRequest(op)* or to execute *Receive(r)*. *GenerateRequest(op)* simply updates the local variables and broadcast the corresponding request to other sites. *Receive(r)* adds r to *WaitingList* and executes all operations of requests that becomes executable thanks to r (relying on *Execute* and *IsExecutable*).

```
1  INITIALIZE():
2  begin
3      ∀i, SReceived[i] = 0          // State Vector of
       received operations
4      (SiteId, Obj, OpCount, WaitingList) = (n, o, 1, {})
5  end

1  GENERATEREQUEST(op): // User emit operation
2  begin
3      Let r = (op, SiteId : OpCount)
4      if isExecutable(r) then
5          OpCount = OpCount + 1
6          t = op(t)                 // Apply operation
7          broadCast r to other participant.
8  end

1  RECEIVE(r): // This function is executed when
   a request is received
2  begin
3      WaitingList = WaitingList ∪ r
4      forall r ∈ WaitingList | isExecutable(r) do
5          execute(r).    // execute all executable
           request
6  end

1  ISEXECUTABLE(r):   // Check that request r is
   executable
2  begin
3      Let r = (op, #Site : #Op)
         // Check that the previous operation on
       same site has been executed
4      if #Site ≠ SiteId ∧ SReceived[#Site] ≠ #Op − 1 then
5          return false
       // Check all dependencies was executed
6      for (nSite : cSite) ∈ dependancesOf(r) do
7          if SReceived[nSite] < cSite then
8              return false
9      return true
10 end

1  EXECUTE(r):              // Execute a request r
2  begin
3      r = (op, #Site : #Op)
4      StateReceived[#Site] = #Op     // Update state
       vector
5      WaitingList = WaitingList/r    // remove r from
       waiting list
6      t = op(t)                 // Applies a operation
7  end
```

Figure 1: The Concurrent Editing Algorithm

The convergence property states that each site has the same copy t of the shared document after all opera-

tions have been received and executed by each site. Firstly, we show that requests are executed in a sequence that respects the dependence relation.

Proposition 2 *Let $op_1^s, \ldots, op_n^s$ be the sequence of operations generated by site s using* GenerateRequest . *Then the operation count associated to op_i^s is i and $op_i^s \succ_s op_j^s$ implies $i < j$.*

PROOF. The first fact is obvious since *OpCount* is incremented by 1 at each creation of an executable request, starting from 0. Line 6 to 9 of *isExecutable(r=(op,#Site,#Op))* tests that each operation op', issued by site *nSite* with operation number *cSite*, which is dependent of *op* contained in *r* has been executed. This is ensured by returning false if $SReceived[nSite] < cSite$. □

Proposition 3 *Let s, s' be two distinct sites. Let $op_1^s, \ldots, op_n^s$ be the sequence of operations generated by s using* GenerateRequest. *Let $op_1^{s'}, \ldots, op_m^{s'}$ be the sequence of operations executed by s' using* GenerateRequest *or* Receive. *If $op_{j_i}^{s'}$ is the execution of op_i^s (from s) by s' then the sequence $op_{j_1}^{s'}, \ldots, op_{j_n}^{s'}$ satisfies $j_1 < j_2 < \ldots < j_n$ (i.e. the execution order on s' respects the creation order on s, hence the dependence relation).*

PROOF. Before any execution of an operation (line 6 of *GenerateRequest* or line 5 of *Receive*) a call to *isExecutable* is performed. The first step of this function returns false for an operation of site *s* numbered *n* if the operation of site *s* numbered $n-1$ has not been executed. Therefore the execution order of the operations op_i^s respects their creation order. Since the creation order respects the dependence relation, we are done. □

Proposition 4 *The algorithm* FCedit *is convergent if the set of operations is independent.*

PROOF. Let $[op_1; \ldots; op_m]$ by the sequence executed on site *s*. We prove that $[op_1; \ldots; op_m]$ is a linearization of the partial order defined by $\succ_s$ on $\{op_1, \ldots, op_m\}$.

Let op_i and op_j such that op_i and op_j have been generated by the same site s'. The subsequence $[op_{j_1}; \ldots; op_{j_l}]$ corresponding to the operations received from site s' is such that $op_{j_k} \succ_s op_{j_{k'}}$ implies $j_k < j_{k'}$ (by proposition 3).

Let op_i and op_j such that op_i has been generated by s' and op_j has been generated by s''. If $op_i \succ_s op_j$, the function *isExecutable* called on the request $r = (op_j, \ldots)$ before executing *r* on site *s* checks that op_i has been executed on site *s* (line 6 to 9 of *isExecutable*). Therefore we get that $i < j$.

Therefore $[op_1; \ldots; op_m]$ is a linearization of the partial order induced by $\succ_s$ on $\{op_1, \ldots, op_m\}$. Since each site executes a linearization of the same partial order, proposition 1 yields that each site computes the same value for the shared document.

□

4 Conflict free operations for XML Trees

The basics editing operations on trees are insertion, deletion or relabeling of a node. Actually, since we consider edge labelled trees instead of node labelled trees, insertion and deletion are performed on edges instead of nodes. Firstly, we consider unordered trees, and we show in section 4.4 how to reestablish the ordering between edges, which allows to get a data-structure corresponding to XML trees.

4.1 Data Structure

The information stored in nodes (or edges in our case) can be described as a word on some finite alphabet Σ. To get a independent set of operations containing relabeling, we must have a much more complex labeling that we describe now.

The set of identifiers *ID***.** Each site is uniquely designated by its identifier which is a natural number (IP numbers could be used as well). The set of identifier is the set *ID* of pairs $((SiteNumber : NbOpns))$ where $NbOpns \in Nat$ is denotes some numbering of operations on this site.

The set of labels $\mathcal{L}$**.** A label is a pair (l, id) where $id \in ID$ and l is a triple (lab, id', dep) with $lab \in \Sigma_L^*$ with Σ_L a finite alphabet, $id' \in ID$, $dep \in \mathcal{N}$ (expressing a level of dependence).

Trees. Trees are defined by the grammar

$$T \ni t ::= \{\ \} \mid \{n_1(t_1), \ldots, n_p(t_p)\} \ \ \textit{where } n_i = (l_i, id_i) \in \mathcal{L}, t_i \in T$$

where **each** id_i **occurs once in** t.

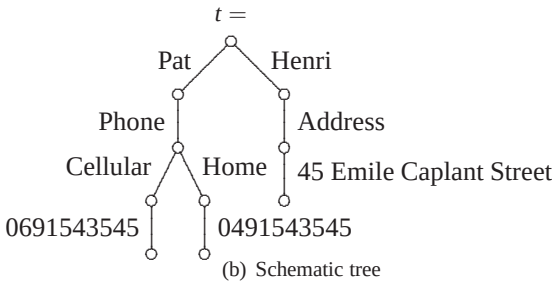
The uniqueness of labels is guaranteed by the fact that $id_i = ((SiteNumber : NbOpns))$ states that the edge has been created by operation *NbOpns* of site *SiteNumber*.

Trees are unordered i.e. $\{n_1(t_1), \ldots, n_p(t_p)\}$ is identified with $\{n_{\sigma(1)}(t_{\sigma(1)}), \ldots, n_{\sigma(p)}(t_{\sigma(p)})\}$ for any permutation of $\{1, \ldots, n\}$.

Example. We give an XML document and a tree that may represent this document as the result of some editing process.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <Pat>
3    <Phone>
4      <Cellular>
5        0691543545
6      </Cellular>
7      <Home>
8        0491543545
9      </Home>
10   </Phone>
11 </Pat>
12 <Henri>
13   <Adress>
14     45 Emile Caplant Street
15   </Adress>
16 </Henri>
```

(a) XML Document



(b) Schematic tree

Figure 2: Document

$$t = \left\{ \begin{array}{l} ((Pat,(1:3),2),(1:1))\left(\left\{ \begin{array}{l} ((Phone,(3:4),5),(2:1))\left(\left\{ \begin{array}{l} ((Home,(3:2),1)(3:1))(\{((0491543545,(4:2),1),(4:1))(\{\})\}) \\ ((Cellular,(5:2),3),(5:1))(\{((0691543545,(6:2),1),(6:1))(\{\})\}) \end{array} \right\}\right) \end{array} \right\}\right) \\ ((Henri,(2:3),1),(2:2))(\{((Address,(3:5),2),(3:2))(\{((45\ Emile\ Caplant\ Street,(4:9),5),(4:2))(\{\})\})\}) \end{array} \right\}$$

4.2 Editing Operations

We extend the set Σ_L by a symbol *NoValue* that states that a label is not yet set.

Adding an edge. The operation $Add(id_p, id)$ with $id_p \neq id$ adds an edge labelled by (l, id) with $l = (NoValue, id, 0)$ under edge labelled $(\ldots, id_p)$. When id_p doesn't occur, the tree is not modified. It is formally defined by:

$$Add(id_p, id)(\{\ \}) = \{\ \}$$

$$Add(id_p, id)(\{n_1(t_1), \ldots, (l_i, id_i)(t_i), \ldots, n_p(t_p)\}) = \{n_1(t_1), \ldots, (l_i, id_i)(t_i \cup ((NoValue, id, 0), id)(\{\}) \ldots n_p(t_p)\} \quad if\ id_p = id_i$$

$$Add(id_p, id)(\{n_1(t_1), \ldots, n_p(t_p)\}) = \{n_1(Add(id_p, id)(t_1)), \ldots, n_p(Add(id_p, id)(t_p))\} \quad if\ n_i = (l_i, id_i)\ with\ id_i \neq id_p\ for\ i = 1, \ldots, n$$

Deleting a subtree. The operation $Del(id)$ deletes the whole subtree corresponding to the unique edge labelled by $(\ldots, id)$ (including this edge). When id doesn't occur, the tree is not modified. It is formally defined by:

$$Del(id)(\{\ \}) = \{\ \}$$

$$Del(id)(\{n_1(t_1), \ldots, (l_i, id_i)(t_i), \ldots, n_p(t_p)\}) = \{n_1(t_1), \ldots, n_{i-1}(t_i), n_{i+1}(t_{i+1}), \ldots n_p(t_p)\} \quad if\ id = id_i$$

$$Del(id)(\{n_1(t_1), \ldots, n_p(t_p)\}) = \{n_1(Del(id)(t_1)), \ldots, n_p(Del(id)(t_p))\} \quad if\ n_i = (l_i, id_i)\ with\ id_i \neq id\ \ for\ i = 1, \ldots, n$$

Changing a label. $ChLab(id_e, id_{op}, dep, L)$ with $id_e, id_{op} \in ID, dep \in \mathcal{N}, L \in \Sigma_L$ replaces the label (l_e, id_e) of the edge identified by $(\ldots, id_e)$ by (L, id_{op}, v) depending on some relations on dependencies. It is defined

formally by:

$$ChLab(id_e, id_{op}, dep, L)(\{n_1(t_1), \ldots (l_e, id_e)(t_e), \ldots n_p(t_p)\}) = \{n_1(t_1), \ldots (l'_e, id_e)(t_e), \ldots, n_p(t_p)$$
$$\text{where } l_e = (L_e, id_e, dep_e) \text{ and } l'_e = \begin{cases} (L, id_{op}, dep), \text{ if } dep_e > dep \text{ or else } dep = dep_e \text{ and } id_{op} < id_{lbl} \\ l_e, \text{ otherwise} \end{cases}$$
$$ChLab(id_e, id_{op}, dep, L)(\{n_1(t_1), \ldots, n_p(t_p)\}) = (\{n_1(ChLab(id_e, id_{op}, dep, L)(t_1)) \ldots n_p(ChLab(id_e, id_{op}, dep, L)(t_p))\})$$
$$\text{if } n_i = (l_i, id_i) \text{ with } id_i \neq id_e \text{ for } i = 1, \ldots, p$$

4.3 Semantic Dependence

Let the set of operations be $Op = \{Add(id, id'), Del(id), ChLab(id, id', dep, L) \mid id, id' \in ID, dep \in \mathcal{N}, L \in \Sigma_L^*\}$. The dependence relation $\succ_s$ is defined as follows:

- $Add(id, id_p) \succ_s Del(id)$: an edge can be deleted only if it has been created.
- $Add(id_p, id_p{}') \succ_s Add(id, id_p)$: adding edge id under edge id_p requires that edge id_p has been created.
- $Add(id, id_p) \succ_s ChLab(id, id_{op}, dep, L)$: changing the labeling of edge id requires that edge id has been created.

This allows to compute the set of identifiers depending from an operation:

$$dependencesOf(op) = \begin{cases} id_p \text{ for } op = Add(id_p, id) \\ id \text{ for } op = Del(id) \\ id \text{ for } op = ChLab(id, id_{op}, depLvl, lbl) \end{cases}$$

Proposition 5 *The set $(Op, \succ_s)$ is an independent set of operations.*

PROOF. We prove that if $op_1 \parallel_s op_2$ then $[op_1, op_2](t) = [op_2, op_1](t)$ by a case analysis on all possible pairs op_1, op_2.

1. $op_1 = Add(id_1, id_{p_1})$
   (a) $op_2 = Add(id_2, id_{p_2})$
   - $id_{p_1} = id_2$ or $id_{p_2} = id_1$ there for respectively $op_1 \succ_s op_2$ or $op_2 \succ_s op_1$.
   - else we can insert a edge before another independently of order the result will be same as a set.

   (b) $op_2 = Del(id_2)$
   - $id_2 = id_{p_1}$ or id_{p_1} is in subtree id_2:
     let t a tree. $t_1 = Del(id_2)(t)$ by definition id_p is deleted.
     $Add(id_1, id_{p_1}) = t_1$. $t_2 = Add(id_2, id_{p_1})(t)$ and $Del(id_2)(t_2) = t_1$ because a subtree are erased.
   - $id_2 = id_1$: because $Add(id_1, id_{p_1}) \succ_s del(id_1)$.
   - other : the edge id_1 has been created and id_2 has been deleted whatever order.

   (c) $op_2 = ChLabel(id_2, id_{op2}, dep_2, lbl_2)$
   - $id_2 = id_1$: the edge be created before renamed because $Add(id_1, id_{p_1}) \succ_s ChLabel(id_1, id_{op2}, dep_2, lbl_2)$.
   - other, the add have no effect on ChLabel and vice versa. ◇

2. $op_1 = Del(id_1)$
   (a) $op_2 = Add(id_2, id_{p_2})$: It's 1b case.
   (b) $op_2 = Del(id_2)$ If id_1 is a subtree id_2 then $[del(id_1), del(id_2)](t)$ there are no edge to delete with $del(id_1)$ because it was deleted with $del(id_2)$. And $[del(id_2), del(id_1)](t)$ the the edge and subedge of id_1 were deleted at first time and id_2 with id_1 was deleted too. else two subtree are distinct .
   (c) $op_2 = ChLabel(id_2, id_{op2}, dep_2, lbl_2)$

- $id_1 = id_2$
  Let $t' = del(id_1)(t)$. $Chlabel(id_1, id_{op2}, dep_2, lbl_2)(t') = t'$ because id_1 is not present in t'.
  $del(id_1)(Chlabel(id_1, id_{op2}, dep_2, lbl_2)(t)) = t'$ because id_1 and it subtree was deleted. Whatever her label.
- Other : there are no problems.

◇

3. $op_1 = Chlabel(id_1, id_{op_1}, dep_1, lbl_1)$

   (a) $op_2 = Add(id_2, id_{p_2})$: It's 1c case.

   (b) $op_2 = Del(id_2)$: It's 2c case.

   (c) $op_2 = ChLabel(id_2, id_{op2}, dep_2, lbl_2)$:

   - $id_1 \neq id_2$: The edge be different.
   - $id_1 = id_2$
     - $dep_1 < dep_2$ let $t_1 = op_1(op_2(t))^{(1)}$
       let $t_2 = op_2(op_1(t))^{(2)}$
       In $^{(1)}$ the label of id_1 is lbl_2 and not changed by op_1 (definition). in $^{(2)}$ the label of id_1 is lbl_1 and changed by op_2 to lbl_2 (definition).
       therefore $t_1 = t_2$.
     - $dep_2 < dep_1$: idem with number of label are inverted.
     - $dep_1 = dep_2$ if $id_{op_1} < id_{op2}$ same of $dep_1 < dep_2$
       else same of $dep_2 < dep_1$
       By definition $id_{op_1} \neq id_{op2}$ ◇

□

4.4 Ordered Trees

The previous editing process is defined on unordered trees when XML documents are ordered trees. To make the algorithm work in this case, we enrich the labeling of edges with an ordering information. This shows that our approach works in this general case. The properties required on the ordering information are:

- The ordering of labels must be a total order
- The ordering is the same for each site
- Insertion can be done between two consecutive edges, before the smallest edge and after the largest edge.

The ordering that we design enjoys all these properties. To each edge corresponding to some identifier *id* we associate a word on some finite alphabet Σ such that two distinct edges corresponds to distinct words.

Let $\Sigma_0 = \{a_1, \ldots, a_n\}$ a finite alphabet such that there is a injective mapping ϕ from ID into Σ_0^*. For instance, to a pair $((s : n))$ with s a site number, n an operation number, we can associate a word $dec(s) \cdot dec(n)$ on the alphabet $\{0, 1, \ldots, 9\} \cup \{\cdot\}$ with $dec(x)$ the representation of x in base 10.

We extend Σ_0 by the letter # used as a separator and $\perp$ used as a minimal element, yielding a alphabet Σ. The ordering on letters is $\perp \leq \# \leq a_1 \ldots < a_n$. The lexicographic ordering on words of Σ^* induced by the ordering of letters is a total ordering.

The labeling of an edge e corresponding to the identifier id_e is enriched by a new field $p_e \in (\Sigma_0 \cup \{\perp; \#\})^*$ and we associate to e the word $w_e = p_e \# \phi(id_e)$. The $\# \phi(id_e)$ part is added to guarantee that distinct edges are associated to distinct words.

Proposition 6 *The ordering on edges defined by $e \prec e'$ iff $w_e = p_e \# \phi(id_e) \ll w_f = p_f \# \phi(id_f)$ is a total ordering on edges.*

PROOF. Since distinct edges have distinct identifier, the function ϕ is injective and $\#\phi(id_e)$ is the smallest suffix of w_e containing only one occurrence of #, then the words associated to distinct edges are distinct. This proves the proposition since $\ll$ is a total ordering on words. □

Example. Let e, f be edges identified by $id_e = (1,10)$ and $id_f = (2,1)$. Let $\phi(id_e) = 1.10$ and $\phi(id_f) = 2.1$. Let the priority of e be 12 and the priority of f be 211. The ordering on digit is $'i' <' j'$ if $i < j$ and $. <' i'$. Since $11\#1.10 \ll 211\#2.1$, we get that edge e precedes edge f in the tree.

Let $\mathcal{W}$ be the set of words of the form $w_p\#w_{id}$ with $w_p \in \Sigma^*$, $w_{id} \in \phi(ID) \subseteq \Sigma_0^*$.

Proposition 7 *Let $w, w' \in \mathcal{W}$ such that $w \ll w'$.*

(i) There exists a computable $w'' \in \mathcal{W}$ such that $w \ll w''$ and $w'' \ll w'$.

(ii) There exists $w_m, w_M \in \mathcal{W}$ such that $w_m \ll w$ and $w' \ll w_M$.

PROOF. Let $s[k]$ denote the k^{th} letter of a word s and let $|s|$ denote the length of the word s.

(i) Let $w = w_p\#w_i \ll w' = w_{p'}\#w'_i$. We construct w'' such that $w \ll w'' \ll w'$. Let j be the minimal integer such that $w[j] < w'[j]$.

Case 1. $j < length(w'_p\#w'_i)$. Let $w_{p''}$ such that $|w''_p| = |w'_p\#w'_i|$ and $w''_p[k] = w'_p[k]$ for $k = 1, \ldots, j$ and $w''_p[k] = \perp$ for $j < k \leq length(w''_p)$. Given any $w''_i = \phi(id)$ for some id, by construction the word $w'' = w''_p\#w''_i$ is such that $w \ll w'' < w'$.

Case 2. $j = length(w'_p\#w'_i)$. Let $w_{p''} = w_p\#w_i\#$. Given any $w''_i = \phi(id)$ for some id, by construction the word $w'' = w''_p\#w''_i$ is such that $w \ll w'' < w'$.

(ii) Let $w = w_p\#w_i < w' = w_{p'}\#w'_i$. We construct w_m such that $w_m \ll w$. Let $w^m_p[k] = \perp$ for $i = 1, \ldots, length(w_p) + 1$. Given any $w^m_i = \phi(id)$ for some id, by construction the word $w_m = w^m_p\#w^m_i$ is such that $w_m < w$. The same construction works to get w_M such that $w' \ll w_M$ (use a_n instead of $\perp$).

□

An updated set of operations. The data structure is slightly modified since the labels are now elements (l, id) with $id \in ID$ and l a tuple $(lab, id', dep, p) \in \Sigma_L^*, id' \in ID, dep \in \mathcal{N}, p \in \mathcal{W}$. The field p combined with the identifier id is used to order the edges arising from the same node, therefore the data structure is similar to semi-structured documents.

The *Add* and *ChLab* operations must be slightly modified to handle the new field p, which simply amounts to considering a different set of labels. The set of dependence between operation is the same as before and we have:

Proposition 8 *The set $(Op, \succ_s)$ is an independent set of operations.*

Therefore our collaborative editing algorithms works for ordered trees, i.e. XML trees.

5 Experiment and Future Works

We have implemented the algorithm and the data structure for XML trees in java (including the ordering information) on a Mac with a 2.53GHz processor.

The data structure *tree* is composed of *edges*. Each edge have the following fields :

- a field for storing its identifier (which is unique).
- a field for storing the sons (which are edges).
- a field for storing its ancestor (which is an edge).

A tree is identified as a some edge (the root). Access to an edge having some identifier is done using a hash-table with identifier as key. The initial document is composed by only one edge: the root with like identifier $0:0$. Applying an operation op on the tree is performed by the function $do : Tree \times Op \longmapsto Tree$.

The implement of *do* is straightforward. For instance $do(Add(id_f, id), tree)$:

(i) creates a new edge with identifier id.

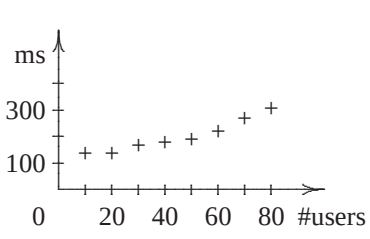
(ii) asks the hash-table to get the father edge id_f

(iii) stores the father reference.

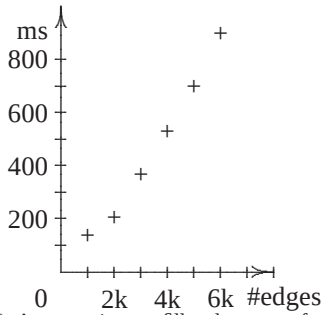
(iv) adds new edge into the father list.

(v) adds new edge references in the hash-table.

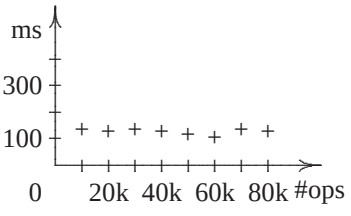
The P2P framework is simulated by random shuffling of the messages that are broadcast. The results obtained with our prototype are given in Figure 3.



(a) Average time computer on function of number of user for 1000 operations



(b) Average time to fill a document for 20 users on all Site



(c) Average computing time to execute 10 000 operations function of the total number of operation performed

Figure 3: Prototype statistic

The reader can see that execution time is almost linear. Furthermore memory consumption (not shown here) is directly related to the size of the document (since we use no history file when for GOTO has a quadratic complexity).

Future works: We plan to extend this word by adding type information like DTD or XML schemas which are used to ensure that XML documents comply with for general structure. The second main extension that we investigate is the ability to *undo* some operations, which may require a limited use of an history file to recover missing information (needed for instance to recover a deleted tree).

REFERENCE

[1] DAVIS, A., SUN, C., AND LU, J. Generalizing operational transformation to the standard general markup language. In *CSCW '02: Proceedings of the 2002 ACM conference on Computer supported cooperative work* (New York, NY, USA, 2002), ACM, pp. 58–67.

[2] DU, L., AND RUI, L. Preserving operation effects relation in group editors. In *CSCW '04: Proceedings of the 2004 ACM conference on Computer supported cooperative work* (New York, NY, USA, 2004), ACM, pp. 457–466.

[3] ELLIS, C. A., AND GIBBS, S. J. Concurrency control in groupware systems. In *SIGMOD Conference* (1989), vol. 18, pp. 399–407.

[4] FOSTER, J., GREENWALD, M., KIRKEGAARD, C., PIERCE, B. C., AND SCHMITT, A. Exploiting schemas in data synchronization. *J. of Computer and System Sciences 73*, 4 (2007).

[5] IGNAT, C., AND NORRIE, M. Tree-based Model Algorithm for Maintaining Consistency in Real-time Collaborative Editing Systems. *Fourth International Workshop on Collaborative Editing, CSCW 2002, IEEE Distributed Systems online* (November 2002).

[6] IGNAT, C., AND NORRIE, M. Customisable Collaborative Editing Supporting the Work Processes of Organisations. *Computers in Industry 57*, 8-9 (December 2006), 758–767.

[7] IMINE, A. *Conception Formelle d'Algorithmes de Réplication Optimiste. Vers l'Edition Collaborative dans les Réseaux Pair-à-Pair*. PhD thesis, Université Henri Poincaré, Nancy, décembre 2006.

[8] IMINE, A., MOLLI, P., OSTER, G., AND RUSINOWITCH, M. Proving correctness of transformation functions in real-time groupware. In *8th European Conference of Computer-supported Cooperative Work* (2003).

[9] KERMARREC, A., ROWSTRON, A., SHAPIRO, M., AND DRUSCHEL, P. The icecube approach to the reconciliation of divergent replicas. In *PODC '01: Proceedings of the twentieth annual ACM symposium on Principles of distributed computing* (New York, NY, USA, 2001), ACM, pp. 210–218.

[10] LI, D., AND LI, R. Ensuring content intention consistency in real-time group editors. In *24th International Conference on Distributed Com- puting Systems* (2004), IEEE Computer Society.

[11] OSTER, G., SKAF-MOLLI, H., MOLLI, P., AND NAJA-JAZZAR, H. Supporting Collaborative Writing of XML Documents. In *Proceedings of the International Conference on Enterprise Information Systems: Software Agents and Internet Computing - ICEIS 2007* (Funchal, Madeira, Portugal, jun 2007), pp. 335–342.

[12] RESSEL, M., NITSCHE-RUHLAND, D., AND GUNZENHÄUSER, R. An integrating, transformation-oriented approach to concurrency control and undo in group editors. In *CSCW '96: Proceedings of the 1996 ACM conference on Computer supported cooperative work* (New York, NY, USA, 1996), ACM, pp. 288–297.

[13] SULEIMAN, M., CART, M., AND FERRIÉ, J. Serialization of concurrent operations in a distributed collaborative environment. In *GROUP '97: Proceedings of the international ACM SIGGROUP conference on Supporting group work* (New York, NY, USA, 1997), ACM, pp. 435–445.

[14] SUN, C., AND ELLIS, C. Operational transformation in real-time group editors: issues, algorithms, and achievements. In *CSCW '98: Proceedings of the 1998 ACM conference on Computer supported cooperative work* (New York, NY, USA, 1998), ACM, pp. 59–68.